and Other Implementations-titleGNU bc and Other Implementations

bc

an arbitrary precision calculator language
version 1.06

Philip A. Nelson

This manual documents `bc`, an arbitrary precision calculator language.
This manual is part of GNU `bc`.

You may contact the author by: e-mail: phil@cs.wwu.edu
us-mail: Philip A. Nelson
Computer Science Department, 9062
Western Washington University
Bellingham, WA 98226-9062

Short Contents

Table of Contents

1 Introduction

1.1 Description

`bc` [-hlwsqv] [long-options] [*file ...*]

`bc` is a language that supports arbitrary precision numbers with interactive execution of statements. There are some similarities in the syntax to the C programming language. A standard math library is available by command line option. If requested, the math library is defined before processing any files. `bc` starts by processing code from all the files listed on the command line in the order listed. After all files have been processed, `bc` reads from the standard input. All code is executed as it is read. (If a file contains a command to halt the processor, `bc` will never read from the standard input.)

This version of `bc` contains several extensions beyond traditional `bc` implementations and the POSIX draft standard. Command line options can cause these extensions to print a warning or to be rejected. This document describes the language accepted by this processor. Extensions will be identified as such.

The author would like to thank Steve Sommars (`Steve.Sommars@att.com`) for his extensive help in testing the implementation. Many great suggestions were given. This is a much better product due to his involvement.

Email bug reports to `bug-bc@gnu.org`. Be sure to include the word "bc" somewhere in the "Subject:" field.

1.2 Command Line Options

`bc` takes the following options from the command line:

`-h, --help`
: Print the usage and exit.

`-l, --mathlib`
: Define the standard math library.

`-w, --warn`
: Give warnings for extensions to POSIX `bc`.

`-s, --standard`
: Process exactly the POSIX `bc` language.

`-q, --quiet`
: Do not print the normal GNU `bc` welcome.

`-v, --version`
: Print the version number and copyright and quit.

2 Basic Elements

2.1 Numbers

The most basic element in `bc` is the number. Numbers are arbitrary precision numbers. This precision is both in the integer part and the fractional part. All numbers are represented internally in decimal and all computation is done in decimal. (This version truncates results from divide and multiply operations.) There are two attributes of numbers, the length and the scale. The length is the total number of significant decimal digits in a number and the scale is the total number of decimal digits after the decimal point. For example, .000001 has a length of 6 and scale of 6, while 1935.000 has a length of 7 and a scale of 3.

2.2 Variables

Numbers are stored in two types of variables, simple variables and arrays. Both simple variables and array variables are named. Names begin with a letter followed by any number of letters, digits and underscores. All letters must be lower case. (Full alphanumeric names are an extension. In POSIX `bc` all names are a single lower case letter.) The type of variable is clear by the context because all array variable names will be followed by brackets ([ ]).

There are four special variables, *scale*, *ibase*, *obase*, and *last*. *scale* defines how some operations use digits after the decimal point. The default value of *scale* is 0. *ibase* and *obase* define the conversion base for input and output numbers. The default for both input and output is base 10. *last* (an extension) is a variable that has the value of the last printed number. These will be discussed in further detail where appropriate. All of these variables may have values assigned to them as well as used in expressions.

2.3 Comments

Comments in `bc` start with the characters `/*` and end with the characters `*/`. Comments may start anywhere and appear as a single space in the input. (This causes comments to delimit other input items. For example, a comment can not be found in the middle of a variable name.) Comments include any newlines (end of line) between the start and the end of the comment.

To support the use of scripts for `bc`, a single line comment has been added as an extension. A single line comment starts at a `#` character and continues to the next end of the line. The end of line character is not part of the comment and is processed normally.

3 Expressions

3.1 About Expressions and Special Variables

The numbers are manipulated by expressions and statements. Since the language was designed to be interactive, statements and expressions are executed as soon as possible. There is no main program. Instead, code is executed as it is encountered. (Functions, discussed in detail later, are defined when encountered.)

A simple expression is just a constant. `bc` converts constants into internal decimal numbers using the current input base, specified by the variable *ibase*. (There is an exception in functions.) The legal values of *ibase* are 2 through 16. Assigning a value outside this range to *ibase* will result in a value of 2 or 16. Input numbers may contain the characters 0-9 and A-F. (Note: They must be capitals. Lower case letters are variable names.) Single digit numbers always have the value of the digit regardless of the value of *ibase*. (i.e. A = 10.) For multi-digit numbers, `bc` changes all input digits greater or equal to *ibase* to the value of *ibase*-1. This makes the number `FFF` always be the largest 3 digit number of the input base.

Full expressions are similar to many other high level languages. Since there is only one kind of number, there are no rules for mixing types. Instead, there are rules on the scale of expressions. Every expression has a scale. This is derived from the scale of original numbers, the operation performed and in many cases, the value of the variable *scale*. Legal values of the variable *scale* are 0 to the maximum number representable by a C integer.

3.2 Basic Expressions

In the following descriptions of legal expressions, "expr" refers to a complete expression and "*var*" refers to a simple or an array variable. A simple variable is just a

name

and an array variable is specified as

name[*expr*]

Unless specifically mentioned the scale of the result is the maximum scale of the expressions involved.

`- expr` The result is the negation of the expression.

`++ var` The variable is incremented by one and the new value is the result of the expression.

`-- var` The variable is decremented by one and the new value is the result of the expression.

`var ++` The result of the expression is the value of the variable and then the variable is incremented by one.

`var --` The result of the expression is the value of the variable and then the variable is decremented by one.

`expr + expr`
The result of the expression is the sum of the two expressions.

`expr - expr`
The result of the expression is the difference of the two expressions.

`expr * expr`
The result of the expression is the product of the two expressions.

`expr / expr`
The result of the expression is the quotient of the two expressions. The scale of the result is the value of the variable `scale`

`expr % expr`
The result of the expression is the "remainder" and it is computed in the following way. To compute a%b, first a/b is computed to *scale* digits. That result is used to compute a-(a/b)*b to the scale of the maximum of *scale*+scale(b) and scale(a). If *scale* is set to zero and both expressions are integers this expression is the integer remainder function.

`expr ^ expr`
The result of the expression is the value of the first raised to the second. The second expression must be an integer. (If the second expression is not an integer, a warning is generated and the expression is truncated to get an integer value.) The scale of the result is *scale* if the exponent is negative. If the exponent is positive the scale of the result is the minimum of the scale of the first expression times the value of the exponent and the maximum of *scale* and the scale of the first expression. (e.g. scale(a^b) = min(scale(a)*b, max(*scale*, scale(a))).) It should be noted that expr^0 will always return the value of 1.

`( expr )` This alters the standard precedence to force the evaluation of the expression.

`var = expr`
The variable is assigned the value of the expression.

`var <op>= expr`
This is equivalent to "*var* = *var* <op> expr" with the exception that the "*var*" part is evaluated only once. This can make a difference if "*var*" is an array.

3.3 Relational Expressions

Relational expressions are a special kind of expression that always evaluate to 0 or 1, 0 if the relation is false and 1 if the relation is true. These may appear in any legal expression. (POSIX `bc` requires that relational expressions are used only in `if`, `while`, and `for` statements and that only one relational test may be done in them.) The relational operators are

`expr1 < expr2`
: The result is 1 if expr1 is strictly less than expr2.

`expr1 <= expr2`
: The result is 1 if expr1 is less than or equal to expr2.

`expr1 > expr2`
: The result is 1 if expr1 is strictly greater than expr2.

`expr1 >= expr2`
: The result is 1 if expr1 is greater than or equal to expr2.

`expr1 == expr2`
: The result is 1 if expr1 is equal to expr2.

`expr1 != expr2`
: The result is 1 if expr1 is not equal to expr2.

3.4 Boolean Expressions

Boolean operations are also legal. (POSIX `bc` does NOT have boolean operations). The result of all boolean operations are 0 and 1 (for false and true) as in relational expressions. The boolean operators are:

`!expr` The result is 1 if expr is 0.

`expr && expr`
: The result is 1 if both expressions are non-zero.

`expr || expr`
: The result is 1 if either expression is non-zero.

3.5 Precedence

The expression precedence is as follows: (lowest to highest)

```
|| operator, left associative
&& operator, left associative
! operator, nonassociative
Relational operators, left associative
Assignment operator, right associative
+ and - operators, left associative
*, / and % operators, left associative
^ operator, right associative
unary - operator, nonassociative
++ and -- operators, nonassociative
```

This precedence was chosen so that POSIX compliant `bc` programs will run correctly. This will cause the use of the relational and logical operators to have some unusual behavior when used with assignment expressions. Consider the expression:

```
a = 3 < 5
```

Most C programmers would assume this would assign the result of "3 < 5" (the value 1) to the variable "a". What this does in `bc` is assign the value 3 to the variable "a" and then compare 3 to 5. It is best to use parentheses when using relational and logical operators with the assignment operators.

3.6 Special Expressions

There are a few more special expressions that are provided in `bc`. These have to do with user-defined functions and standard functions. They all appear as "*name*(*parameters*)". See Chapter 5 [Functions], page 10, for user-defined functions. The standard functions are:

`length ( expression )`
: The value of the length function is the number of significant digits in the expression.

`read ( )`
: The `read` function (an extension) will read a number from the standard input, regardless of where the function occurs. Beware, this can cause problems with the mixing of data and program in the standard input. The best use for this function is in a previously written program that needs input from the user, but never allows program code to be input from the user. The value of the `read` function is the number read from the standard input using the current value of the variable *ibase* for the conversion base.

`scale ( expression )`
: The value of the `scale` function is the number of digits after the decimal point in the expression.

`sqrt ( expression )`
: The value of the `sqrt` function is the square root of the expression. If the expression is negative, a run time error is generated.

4 Statements

Statements (as in most algebraic languages) provide the sequencing of expression evaluation. In `bc` statements are executed "as soon as possible." Execution happens when a newline in encountered and there is one or more complete statements. Due to this immediate execution, newlines are very important in `bc`. In fact, both a semicolon and a newline are used as statement separators. An improperly placed newline will cause a syntax error. Because newlines are statement separators, it is possible to hide a newline by using the backslash character. The sequence "\<nl>", where <nl> is the newline appears to `bc` as whitespace instead of a newline. A statement list is a series of statements separated by semicolons and newlines. The following is a list of `bc` statements and what they do: (Things enclosed in brackets ([ ]) are optional parts of the statement.)

expression
: This statement does one of two things. If the expression starts with "<variable> <assignment> ...", it is considered to be an assignment statement. If the expression is not an assignment statement, the expression is evaluated and printed to the output. After the number is printed, a newline is printed. For example, "a=1" is an assignment statement and "(a=1)" is an expression that has an embedded assignment. All numbers that are printed are printed in the base specified by the variable *obase*. The legal values for *obase* are 2 through BC_BASE_MAX (see ⟨undefined⟩ [Environment Variables], page ⟨undefined⟩). For bases 2 through 16, the usual method of writing numbers is used. For bases greater than 16, `bc` uses a multi-character digit method of printing the numbers where each higher base digit is printed as a base 10 number. The multi-character digits are separated by spaces. Each digit contains the number of characters required to represent the base ten value of "*obase* -1". Since numbers are of arbitrary precision, some numbers may not be printable on a single output line. These long numbers will be split across lines using the "\" as the last character on a line. The maximum number of characters printed per line is 70. Due to the interactive nature of `bc`, printing a number causes the side effect of assigning the printed value to the special variable *last*. This allows the user to recover the last value printed without having to retype the expression that printed the number. Assigning to *last* is legal and will overwrite the last printed value with the assigned value. The newly assigned value will remain until the next number is printed or another value is assigned to *last*. (Some installations may allow the use of a single period (.) which is not part of a number as a short hand notation for for *last*.)

string The string is printed to the output. Strings start with a double quote character and contain all characters until the next double quote character. All characters are taken literally, including any newline. No newline character is printed after the string.

`print` *list* The `print` statement (an extension) provides another method of output. The *list* is a list of strings and expressions separated by commas. Each string or expression is printed in the order of the list. No terminating newline is printed. Expressions are evaluated and their value is printed and assigned to the variable `last`. Strings in the print statement are printed to the output and may contain special characters. Special characters start with the backslash character (\e). The special characters recognized by `bc` are `"a"` (alert or bell), `"b"` (backspace), `"f"` (form feed), `"n"` (newline), `"r"` (carriage return), `"q"` (double quote), `"t"` (tab), and `"\e"` (backslash). Any other character following the backslash will be ignored.

`{` *statement_list* `}`
This is the compound statement. It allows multiple statements to be grouped together for execution.

`if` *(expression) statement1 [*`else` *statement2]*
The if statement evaluates the expression and executes statement1 or statement2 depending on the value of the expression. If the expression is non-zero, statement1 is executed. If statement2 is present and the value of the expression is 0, then statement2 is executed. (The `else` clause is an extension.)

`while` *(expression) statement*
The while statement will execute the statement while the expression is non-zero. It evaluates the expression before each execution of the statement. Termination of the loop is caused by a zero expression value or the execution of a `break` statement.

`for` *([expression1] ; [expression2] ; [expression3]) statement*
The `for` statement controls repeated execution of the statement. *Expression1* is evaluated before the loop. *Expression2* is evaluated before each execution of the statement. If it is non-zero, the statement is evaluated. If it is zero, the loop is terminated. After each execution of the statement, *expression3* is evaluated before the reevaluation of expression2. If *expression1* or *expression3* are missing, nothing is evaluated at the point they would be evaluated. If *expression2* is missing, it is the same as substituting the value 1 for *expression2*. (The optional expressions are an extension. POSIX `bc` requires all three expressions.) The following is equivalent code for the `for` statement:

```
expression1;
```

```
while (expression2) {
   statement;
   expression3;
}
```

`break` This statement causes a forced exit of the most recent enclosing `while` statement or `for` statement.

`continue` The `continue` statement (an extension) causes the most recent enclosing `for` statement to start the next iteration.

`halt` The `halt` statement (an extension) is an executed statement that causes the `bc` processor to quit only when it is executed. For example, "if (0 == 1) halt" will not cause `bc` to terminate because the `halt` is not executed.

`return` Return the value 0 from a function. (See Chapter 5 [Functions], page 10.)

`return` (*expression*)
Return the value of the expression from a function. (See Chapter 5 [Functions], page 10.) As an extension, the parenthesis are not required.

4.1 Pseudo Statements

These statements are not statements in the traditional sense. They are not executed statements. Their function is performed at "compile" time.

`limits` Print the local limits enforced by the local version of `bc`. This is an extension.

`quit` When the `quit` statement is read, the `bc` processor is terminated, regardless of where the `quit` statement is found. For example, "if (0 == 1) quit" will cause `bc` to terminate.

`warranty` Print a longer warranty notice. This is an extension.

5 Functions

Functions provide a method of defining a computation that can be executed later. Functions in `bc` always compute a value and return it to the caller. Function definitions are "dynamic" in the sense that a function is undefined until a definition is encountered in the input. That definition is then used until another definition function for the same name is encountered. The new definition then replaces the older definition. A function is defined as follows:

```
define name ( parameters ) { newline
    auto_list   statement_list }
```

A function call is just an expression of the form "`name (`*parameters*`)`".

Parameters are numbers or arrays (an extension). In the function definition, zero or more parameters are defined by listing their names separated by commas. Numbers are only call by value parameters. Arrays are only call by variable. Arrays are specified in the parameter definition by the notation "*name*`[ ]`". In the function call, actual parameters are full expressions for number parameters. The same notation is used for passing arrays as for defining array parameters. The named array is passed by variable to the function. Since function definitions are dynamic, parameter numbers and types are checked when a function is called. Any mismatch in number or types of parameters will cause a runtime error. A runtime error will also occur for the call to an undefined function.

The *auto_list* is an optional list of variables that are for "local" use. The syntax of the auto list (if present) is "`auto` *name*, ... ;". (The semicolon is optional.) Each *name* is the name of an auto variable. Arrays may be specified by using the same notation as used in parameters. These variables have their values pushed onto a stack at the start of the function. The variables are then initialized to zero and used throughout the execution of the function. At function exit, these variables are popped so that the original value (at the time of the function call) of these variables are restored. The parameters are really auto variables that are initialized to a value provided in the function call. Auto variables are different than traditional local variables because if function A calls function B, B may access function A's auto variables by just using the same name, unless function B has called them auto variables. Due to the fact that auto variables and parameters are pushed onto a stack, `bc` supports recursive functions.

The function body is a list of `bc` statements. Again, statements are separated by semicolons or newlines. Return statements cause the termination of a function and the return of a value. There are two versions of the return statement. The first form, "`return`", returns the value 0 to the calling expression. The second form, "`return` (*expression*)", computes the value of the expression and returns that value to the calling expression. There is an implied "`return` (0)" at the end of every function. This allows a function to terminate and return 0 without an explicit `return` statement.

Functions also change the usage of the variable *ibase*. All constants in the function body will be converted using the value of *ibase* at the time of the function call. Changes of *ibase* will be ignored during the execution of the function except for the standard function `read`, which will always use the current value of *ibase* for conversion of numbers.

As an extension, the format of the definition has been slightly relaxed. The standard requires the opening brace be on the same line as the `define` keyword and all other parts must be on following lines. This version of `bc` will allow any number of newlines before and after the opening brace of the function. For example, the following definitions are legal.

```
define d (n) { return (2*n); }
define d (n)
  { return (2*n); }
```

5.1 Math Library Functions

If `bc` is invoked with the `-l` option, a math library is preloaded and the default *scale* is set to 20. The math functions will calculate their results to the scale set at the time of their call. The math library defines the following functions:

`s (`*x*`)` The sine of *x*, *x* is in radians.

`c (`*x*`)` The cosine of *x*, *x* is in radians.

`a (`*x*`)` The arctangent of *x*, arctangent returns radians.

`l (`*x*`)` The natural logarithm of *x*.

`e (`*x*`)` The exponential function of raising *e* to the value *x*.

`j (`*n*`,`*x*`)` The bessel function of integer order *n* of *x*.

6 Examples

In /bin/sh, the following will assign the value of "pi" to the shell variable *pi*.

```
pi=$(echo "scale=10; 4*a(1)" | bc -l)
```

The following is the definition of the exponential function used in the math library. This function is written in POSIX `bc`.

```
scale = 20

/* Uses the fact that e^x = (e^(x/2))^2
   When x is small enough, we use the series:
     e^x = 1 + x + x^2/2! + x^3/3! + ...
*/

define e(x) {
  auto  a, d, e, f, i, m, v, z

  /* Check the sign of x. */
  if (x<0) {
    m = 1
    x = -x
  }

  /* Precondition x. */
  z = scale;
  scale = 4 + z + .44*x;
  while (x > 1) {
    f += 1;
    x /= 2;
  }

  /* Initialize the variables. */
  v = 1+x
  a = x
  d = 1

  for (i=2; 1; i++) {
    e = (a *= x) / (d *= i)
    if (e == 0) {
      if (f>0) while (f--)  v = v*v;
      scale = z
      if (m) return (1/v);
```

```
        return (v/1);
      }
      v += e
    }
  }
```

The following is code that uses the extended features of bc to implement a simple program for calculating checkbook balances. This program is best kept in a file so that it can be used many times without having to retype it at every use.

```
scale=2
print "\nCheck book program\n!"
print "  Remember, deposits are negative transactions.\n"
print "  Exit by a 0 transaction.\n\n"

print "Initial balance? "; bal = read()
bal /= 1
print "\n"
while (1) {
  "current balance = "; bal
  "transaction? "; trans = read()
  if (trans == 0) break;
  bal -= trans
  bal /= 1
}
quit
```

The following is the definition of the recursive factorial function.

```
define f (x) {
  if (x <= 1) return (1);
  return (f(x-1) * x);
}
```

7 Readline and Libedit Options

GNU `bc` can be compiled (via a configure option) to use the GNU `readline` input editor library or the BSD `libedit` library. This allows the user to do more editing of lines before sending them to `bc`. It also allows for a history of previous lines typed. When this option is selected, `bc` has one more special variable. This special variable, *history* is the number of lines of history retained. A value of -1 means that an unlimited number of history lines are retained. This is the default value. Setting the value of *history* to a positive number restricts the number of history lines to the number given. The value of 0 disables the history feature. For more information, read the user manuals for the GNU `readline`, `history` and BSD `libedit` libraries. One can not enable both `readline` and `libedit` at the same time.

8 GNU bc and Other Implementations

This version of bc was implemented from the POSIX P1003.2/D11 draft and contains several differences and extensions relative to the draft and traditional implementations. It is not implemented in the traditional way using dc. This version is a single process which parses and runs a byte code translation of the program. There is an "undocumented" option (-c) that causes the program to output the byte code to the standard output instead of running it. It was mainly used for debugging the parser and preparing the math library.

A major source of differences is extensions, where a feature is extended to add more functionality and additions, where new features are added. The following is the list of differences and extensions.

LANG environment
: This version does not conform to the POSIX standard in the processing of the LANG environment variable and all environment variables starting with LC_.

names
: Traditional and POSIX bc have single letter names for functions, variables and arrays. They have been extended to be multi-character names that start with a letter and may contain letters, numbers and the underscore character.

Strings
: Strings are not allowed to contain NUL characters. POSIX says all characters must be included in strings.

last
: POSIX bc does not have a \fBlast variable. Some implementations of bc use the period (.) in a similar way.

comparisons
: POSIX bc allows comparisons only in the if statement, the while statement, and the second expression of the for statement. Also, only one relational operation is allowed in each of those statements.

if statement, else clause
: POSIX bc does not have an else clause.

for statement
: POSIX bc requires all expressions to be present in the for statement.

&&, ||, !
: POSIX bc does not have the logical operators.

read function
: POSIX bc does not have a read function.

print statement
: POSIX bc does not have a print statement.

continue statement
: POSIX `bc` does not have a continue statement.

array parameters
: POSIX `bc` does not (currently) support array parameters in full. The POSIX grammar allows for arrays in function definitions, but does not provide a method to specify an array as an actual parameter. (This is most likely an oversight in the grammar.) Traditional implementations of `bc` have only call by value array parameters.

function format
: POSIX `bc` requires the opening brace on the same line as the `define` key word and the `auto` statement on the next line.

=+, =-, =, =/, =%, =^*
: POSIX `bc` does not require these "old style" assignment operators to be defined. This version may allow these "old style" assignments. Use the `limits` statement to see if the installed version supports them. If it does support the "old style" assignment operators, the statement "a =- 1" will decrement `a` by 1 instead of setting `a` to the value -1.

spaces in numbers
: Other implementations of `bc` allow spaces in numbers. For example, "x=1 3" would assign the value 13 to the variable x. The same statement would cause a syntax error in this version of `bc`.

errors and execution
: This implementation varies from other implementations in terms of what code will be executed when syntax and other errors are found in the program. If a syntax error is found in a function definition, error recovery tries to find the beginning of a statement and continue to parse the function. Once a syntax error is found in the function, the function will not be callable and becomes undefined. Syntax errors in the interactive execution code will invalidate the current execution block. The execution block is terminated by an end of line that appears after a complete sequence of statements. For example,

```
a = 1
b = 2
```

has two execution blocks and

```
{ a = 1
  b = 2 }
```

has one execution block. Any runtime error will terminate the execution of the current execution block. A runtime warning will not terminate the current execution block.

Interrupts During an interactive session, the SIGINT signal (usually generated by the control-C character from the terminal) will cause execution of the current execution block to be interrupted. It will display a "runtime" error indicating which function was interrupted. After all runtime structures have been cleaned up, a message will be printed to notify the user that `bc` is ready for more input. All previously defined functions remain defined and the value of all non-auto variables are the value at the point of interruption. All auto variables and function parameters are removed during the clean up process. During a non-interactive session, the SIGINT signal will terminate the entire run of `bc`.

9 Limits

The following are the limits currently in place for this `bc` processor. Some of them may have been changed by an installation. Use the `limits` statement to see the actual values.

`BC_BASE_MAX`
: The maximum output base is currently set at 999. The maximum input base is 16.

`BC_DIM_MAX`
: This is currently an arbitrary limit of 65535 as distributed. Your installation may be different.

`BC_SCALE_MAX`
: The number of digits after the decimal point is limited to INT_MAX digits. Also, the number of digits before the decimal point is limited to INT_MAX digits.

`BC_STRING_MAX`
: The limit on the number of characters in a string is INT_MAX characters.

`exponent`
: The value of the exponent in the raise operation (^) is limited to LONG_MAX.

`multiply`
: The multiply routine may yield incorrect results if a number has more than LONG_MAX / 90 total digits. For 32 bit longs, this number is 23,860,929 digits.

`variable names`
: The current limit on the number of unique names is 32767 for each of simple variables, arrays and functions.

10 Environment Variables

The following environment variables are processed by `bc`:

`POSIXLY_CORRECT`
: This is the same as the -s option (see Section 1.2 [Command Line Options], page 1).

`BC_ENV_ARGS`
: This is another mechanism to get arguments to `bc`. The format is the same as the command line arguments. These arguments are processed first, so any files listed in the environent arguments are processed before any command line argument files. This allows the user to set up `"standard"` options and files to be processed at every invocation of `bc`. The files in the environment variables would typically contain function definitions for functions the user wants defined every time `bc` is run.

`BC_LINE_LENGTH`
: This should be an integer specifing the number of characters in an output line for numbers. This includes the backslash and newline characters for long numbers.